Design and Implementation of OpenXM client server model and common mathematical object format (OpenXM-RFC 100, proposed standard)

— Open message eXchange protocol for Mathematics

Masayuki Noro, Nobuki Takayama*

November 20, 1997 — November 17, 2000,
August 27, 2001, January 20, 2002 (minor change),
See changeLog about changes after this date.

ChangeLog
2004-3-8: We add a new stackmacine command `SM_executeFunctionWithOptionalArgument`.
2005-3-4: Added a description about a byte order negotiation to send floating point numbers. `CMO_64BIT_MACHINE_DOUBLE`

*Department of Mathematics, Kobe University,http://www.math.kobe-u.ac.jp/~taka

Contents

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

1 Introduction

OpenXM is a free, or Open Source, infrastructure for mathematical software systems. It provides methods and protocols for interactive distributed computation and for integrating mathematical software systems. OpenXM package is a set of software systems that support OpenXM protocols. It is currently a collection of software systems `Risa/Asir` [3], `Kan/sm1` [7], `PHC` pack [8] , `GNUPLOT`, `Mathematica` interface, and `OpenMath`/XML [4] translator.

We have been profited from increasing number of mathematical software systems. These are usually "expert" systems in one area of mathematics such as ideals, groups, numbers, polytopes, and so on. They have their own interfaces and data formats, which are fine for intensive users of these systems. However, a unified system will be more convenient for users who want to explore a new area of mathematics with these software systems or users who need these systems only occasionally. It is also wonderful for developpers to have various software components that can be used from the target system.

OpenXM provides not only data representation and communication protocols but also programming guidelines to develop cooperative applications. One will be able to concentrate on developing mathematical algorithms with such guidelines. Our design goals are (1) simpleness, (2) extensibility, (3) easiness of implementation, (4) practicality, and (5) robustness.

We believe that an open integrated system is a future of mathematical software systems. However, it might be a dream without realizability. We want to build a prototype of such an open system by using existing standards, technologies and several mathematical software systems. We want to see how far we can go with this approach.

It is not an obvious problem to consider how mathematical objects are represented and communicated. It may be similar to trying to create new mathematical symbols. We have the decimal notation to represent numbers, the symbol dx to represent a differential, and $\longrightarrow$ to represent a mapping. One should imagine how we are benefited from these notations.

In OpenXM, communication is an exchange of messages. The messages are classified into three types: DATA, COMMAND, and SPECIAL. They are called OX (OpenXM) messages. Among the three types, *OX data messages* wrap mathematical data. We use standards of mathematical data formats such as OpenMath and MP as well as our own data format *CMO* (*Common Mathematical Object format*). Servers, which provide services to other processes, are stack machines. The stack machine is called the *OX stack machine*. Existing mathematical software systems are wrapped with this stack machine. OX stack machines work in the asynchronous mode like X servers. OpenXM servers try to be as quiet as possible. OpenXM server does not send messages to the client unless it is requested to send them.

Our stackmachine architecutre can be used as the lowest level layer to implement other protocols. Emulating RPC or constructing a web server like MCP [9] on our asynchronous OX stack machines are possible.

Our datatype definition is compliant to XML architecture. OX messages can be defined by DTD and can be expressed by XML. We call it OpenXM/XML.

A system xxx complient to the OpenXM protocol is called open xxx. For example Asir complient to the OpenXM protocol is called open Asir, and kan/sm1 complient to the OpenXM protocol is called open sm1.

2 CMO Primitive object

Objects in CMO (Common Mathematical Object format) group Primitive are primitive data such as `int`, `string`. All OpenXM compliant systems should implement all data types in the group Primitive. In this section, as an introduction, we will introduce CMObject (Common Mathematical Object) of the group Primitive without using the Backus-Nauer form.

The canonical name of this group is CMObject/Primitive. In the sequel, `int32` means the signed 32 bit integer expressed by two's complement (internal expressions of `int` of the language C usually use this expression). `byte` means 8 bit data.

In our encoding of the CMO's for TCP/IP, any CMObject consists of a tag and a body:

cmo_tag	cmo_body

`cmo_tag` should be given by a positive `int32`.

The following is a list of tags of CMObject/Primitive. @../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
```

We will explain each object format. Servers and clients do not need to implement all CMO's. However, `CMO_ERROR2`, `CMO_NULL`, `CMO_INT32`, `CMO_STRING`, `CMO_MATHCAP`, `CMO_LIST` are primitive data and all servers and clients have to implement them.

CMObject Error2 is of the form

int32 CMO_ERROR2	*CMObject* ob

It is an object used when a server makes an error. *CMObject* ob carries error informations. The instance ob is a list and, in case of a stream connection like

TCP/IP, the first element must be the serial number of the OX message that caused the error. The serial number is given by the data type Integer32.
Remark: For a historical reason the CMO tag of the error object is named `CMO_ERROR2`. In the next version of OpenXM specification we may rename it `CMO_ERROR`.

CMObject Null has the format

`int32 CMO_NULL`

32 bit integer n is called Integer32 as a CMObject and has the format

`int32 CMO_INT32`	`int32` n

A byte array of the length n is called Datum as a CMObject and has the format

`int32 CMO_DATUM`	`int32` n	`byte` data[0]	`byte` data[1]
$\cdots$	`byte` data[n-1]		

String of n bytes is called Cstring as CMObject and has the format

`int32 CMO_STRING`	`int32` n	`byte` data[0]	`byte` data[1]
$\cdots$	`byte` data[n-1]		

CMObject Mathcap has the format

`int32 CMO_MATHCAP`	*CMObject* ob

ob is a list of which length is more than or equal to three. The first element is a list of OpenXM protocol version number in Integer32, the server name in Cstring, the server version and CPU type in Cstring, and extra informations. The second element is a list of SM tags in Integer 32. The third element is a list of data type tags which the server or the client can understand. The details will be explained in the section on mathcap 5.1.2.

A list of the length m has the form

`int32 CMO_LIST`	`int32` m	`CMObject` ob[0]	$\cdots$	`CMObject` ob[$m-1$]

3 A formal expression of CMO

In the previous section, we have explained the format of CMO's in the Primitive group. In this section, we will introduce CMOexpression which is like the bracket expression of Lisp. We again explain a standard encoding method of CMO, which we have already explained in the previous section, but the explanation is more formal.

Let us define CMOexpression by the extended BNF expression. Symbols in the type writer fonts mean terminals. ":" means a definition. "|" means "or". { X } is a repetition of X of more than or equal to 0 times. [x] stands for X or nothing. By using this notation, CMOexpression is defined as follows.

$$
\begin{array}{rcl}
\text{CMOexpression} & : & (\texttt{cmo\_tag} \ \{ \ \text{expression}\}) \\
\text{expression} & : & \text{CMOexpression} \\
 & & |\ \texttt{int32} \\
 & & |\ \texttt{string} \\
 & & |\ \texttt{byte}
\end{array}
$$

Terminal `int32` is signed 32 bit integer. Terminal `string` is a byte array which usually expresses a string. Terminal `byte` is 8 bit data.

The comma (`,`) may be used to separate each element in CMOexpressions. `cmo_tag` is a constant that starts with `CMO_`.

Let us describe CMO's in the Primitive group. In order to explain the meaning of objects, we may also put variable names to CMOexpressions. The start of comments are denoted by "—".

By using this notation, let us define formally CMObjects in the group Primitive.

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)

$$
\begin{array}{rcl}
\text{Error2} & : & (\texttt{CMO\_ERROR2}, \mathit{CMObject}\ \text{ob}) \\
\text{Null} & : & (\texttt{CMO\_NULL}) \\
\text{Integer32} & : & (\texttt{CMO\_INT32}, \mathit{int}32\ \text{n}) \\
\text{Datum} & : & (\texttt{CMO\_DATUM}, \mathit{int}32\,\text{n}, \mathit{byte}\,\text{data[0]}, \ldots, \mathit{byte}\,\text{data[n-1]}) \\
\text{Cstring} & : & (\texttt{CMO\_STRING}, \mathit{int}32\ \text{n}, \mathit{string}\,\text{s}) \\
\text{Mathcap} & : & (\texttt{CMO\_MATHCAP}, \mathit{CMObject}\,\text{ob}) \\
\text{List} & : & (\texttt{CMO\_LIST}, \mathit{int32}\ \text{m}, \mathit{CMObject}\ \text{ob[0]}, \ldots, \mathit{CMObject}\ \text{ob[m-1]}) \\
 & & \text{— m is the length of the list.}
\end{array}
$$

In the definition of "Cstring", if we decompose "*string* s" into bytes, then "Cstring" should be defined as

$$
\text{Cstring} \quad : \quad (\texttt{CMO\_STRING}, \mathit{int}32\ \text{n}, \mathit{byte}\,\text{s[0]}, \ldots, \mathit{byte}\ \text{s[n-1]})
$$

"Group CMObject/Primitive requires nothing" means that there is no super group to define CMO's in the group Primitive. "Error2, Null, Integer32, Datum, Cstring, Mathcap, List ∈ CMObject/Primitive" means that Error2, Null, Integer32, Datum, Cstring are members of the group CMObject/Primitive.

Let us see examples. 32 bit integer 1234 is expressed as

$$(\texttt{CMO\_INT32}, 1234)$$

The string “Hello” is expressed as

$$(\texttt{CMO\_STRING}, 5, \text{"Hello"})$$

CMO’s are expressed by XML like Open Math ([4]). See example below.

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

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

cmo_string may be expressed as follows.

```
<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>
```

In this case, the DTD for cmo_string is as follows;

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

Let us explain the standard encoding method. All `int32` data are encoded into network byte order 32 bit integers and byte data are encoded as it is.

When we are using a high speed network, the translation from the internal expression of 32 bit integers to network byte order may become a bottle neck. There are experimental data which presents that 90 percents of the transmission time are used for the translation to the network byte order to send `CMO_ZZ` of size 12M bytes. We used a 100Mbps network. In a later section 8.3, we will discuss a protocol to avoid the translation.

The translation between the standard encoding and CMOexpression is easy. For example,

int32 CMO_INT32	int32 1234

is the encoding of the CMOexpression

$$(\texttt{CMO\_INT32}, 1234)$$

(Experimental) CMO and OX packets are complient to XML specification [10]. In order to encode "Attribute" in XML in our binary format, we have a tag:

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

For example, the attibute `font="Times-Roman"` is encoded as

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

All tags except this special CMO tag `CMO_ATTRIBUTE_LIST` are XML tags in the CMO/XML expression.

CMO/XML attributes such as *comment*, *for* are not encoded in the CMO binary expression.

4 Communication model of OpenXM

In our model of computation, mathematical processes proceed a computation by exchanging messages. Each process is a stack machine, which is called an OX stack machine. The following methods are possible to realize communications between mathematical processes.

1. Communication by files.
2. Linking as a subroutine library.
3. TCP/IP streams.
4. Remote Procedure call.
5. JAVA RMI.
6. Multi-thread.
7. PVM library.
8. MPI library.

In OpenXM communication means exchange of messages between processes. A message has the following structure:

`destination`	`origin`	
`extension`	`ox message_tag`	`message_body`

We call it an OX message (OpenXM message object). OX message is the top level message object. The OX messages are classified into three types: DATA, COMMAND, and SPECIAL. They are distinguished by `ox message_tag`. The name of an ox message tag begins with `OX_`. Typical OX message tags are

OX_COMMAND followed by SMobject and OX_DATA followed by CMObject. Each message object also has its tag. For SMobject, the name of a tag begins with SM_. For CMObject, the name of a tag begins with CMO_. An SMobject represents a stack machine command and categorized into several groups such as SMobject/Primitive, SMobject/Basic. The details of SMobjects will be explained in Section 5. We have already explained the Primitive CMObjects. We will describe the Basic CMObjects in Section 10.

4.1 OX Messages

In OpenXM, each process may have a hybrid interface; it may accept and execute not only stack machine commands, but also its original command sequences. We call such a process an OX stack machine. Here we introduce OXexpression and SMexpression to express OX messages and SM objects respectively.

4.1.1 Expressions of OX messages (Lisp like)

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

A comma ',' may be used to separate elements in an expression. OX_tag is a constant which denotes an OX message tag. SM_tag is a constant which denotes an SM command tag. If a sender AAA or a receiver BBB has to be specified, 'From AAA' or 'To BBB' is written before the OXexpression.

For example the following expression means a request to push a CMO string "Hello".

(OX_DATA, (CMO_STRING, 5, "Hello"))

The following expression means a request to execute a local function "hoge".

(OX_DATA, (CMO_STRING, 4, "hoge"))

(OX_COMMAND, SM_executeStringByLocalParser)

In our standard encoding method, each tag is expressed as a 32 bit (4 bytes) integer with the network byte order.

4.1.2 Expression of OX messages (XML)

OX messages can be expressed by XML. The following is an example.

```
<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 Standard encoding of OXexpressions and an implementation by TCP/IP sockets

The logical structure of OX messages are independent of implementations of communication. The OXexpression represents the logical structure. Here we explain an outline of the standard encoding scheme of OXexpression. This encoding scheme is used to implement OpenXM protocols on TCP/IP sockets. In addition, we also explain the control messages to control stack machines.

As the socket connection is peer to peer, `destination` and `origin` are omitted. The `extension` field is placed after the `message_tag` field. The `extension` field consists of the serial number for OX message, which is `int32`. The serial number is used to identify an OX message which caused an error on a server. In the following we regard the `extension` as a component of the `message_tag` field and omit the `extension` field. Thus OX messages are represented as follows.

`ox message_tag`	`message_body`

More precisely it has the following representation.

`ox message_tag, serial number`	`message_body`

As `ox message_tag` the following are provided.
@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
```

Two streams are provided for communication between a client and a server. One is the stream to exchange data and to send stack machine commands. The other is the stream to control stack machines. Messages on the latter stream are called control messages and results of control messages. The sample server implements the above two streams by using two ports on TCP/IP.

The stack machine command message has the following forms:

OX_COMMAND	int32 function_id
message_tag	*message_body*

, (OX_COMMAND, (SM_*))

CMO data message has the following form:

OX_DATA	CMO data
message_tag	*message_body*

, (OX_DATA, *CMObject* data)

The control message has the following form:

OX_COMMAND	int32 function_id

, (OX_COMMAND,(SM_control_*))

The control message is used to interrupt a computation, to invoke debugging threads, or to exit form the debugging mode.

The result of a control message has the following form:

OX_DATA	CMO_INT32	int32 data

, (OX_DATA, *Integer32* n)

`int32 function_id` is the value of a stack machine command. SM tags in SMobject/Primitive and SMobject/Basic and corresponding values are as follows.

@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
```

For example

$$(\text{OX\_COMMAND}, \text{SM\_pops})$$

is encoded as follows.

int32 513	int32 265

The details of the operators are described in Section 5. Names of these constants may be represented by abbreviated forms.

5 OX stack machine

In this section we describe the OX stack machine operators. In the descriptions OX messages are represented by th standard encoding scheme on TCP/IP sockets. In principle, an OX stack machine never sends data to the output stream unless it receives `SM_pop*` commands. Note that the programming style should be different from that for event-driven programming.

5.1 Server stack machine

`oxserver00.c` is implemented as a sample server. If you want to implement you own server, write the following functions and use them instead of those in `nullstackmachine.c`.

5.1.1 Operators in the group SMobject/Primitive

Any OX stack machine has at least one stack.

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

Here `Object` may be local to the system `xxx` wrapped by the stack machine. That is, the server may translate CMObjects into its local objects and push them onto the stack. It is preferable that the composition of such a translation and its inverse is equal to the identity map. The translation scheme is called the *phrase book* of the server and it should be documented for each stack machine. In OpenXM, any message is private to a connection. In future we will provide a content dictionary (CD; see OpenMath [4]) for basic specifications of CMObjects.

In the following, `xxx_` may be omitted if no confusion occurs. As the names of functions and tags are long, one may use abbreviated names. Message packets are represented as follows.

Each field is shown as | data type data |. For example `int32 OX_DATA` denotes a number `OX_DATA` which is represented by a 32 bit integer with the network byte order. If a field is displayed by italic characters, it should be defined elsewhere or its meaning should be clear. For example *String commandName* denotes a local object *commandName* whose data type is String. Note that an object on the stack may have a local data type even if it is represented as CMO.

Any server stack machine has to implement the following operations. For each operation we show the states of the stack before and after the operation. In the figures the rightmost object corresponds to the top of the stack. Only the modified part of the stack are shown.

1. Any server should accept CMObjects in the group CMObject/Primitive. The server pushes such data onto the stack. The following examples show the states of the stack after receiving `CMO_NULL` or `CMO_String` respectively.

   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	$\cdots$	`byte` ssize

   Stack after the request: | *String s* |

   Output: none.

   If the server fails to receive a CMO data,

`int32 OX_DATA`	`int32 CMO_ERROR2`	*CMObject* ob

   is pushed onto the stack. Currently ob is a list
[*Integer32* OX serial number, *Integer32* error code, *CMObject* optional information]

2. `SM_mathcap`

   It requests a server to push the mathcap of the server. The mathcap is similar to the termcap. One can know the server type and the capability of the server from the mathcap.

   @plugin/mathcap.h)

   Request:

`int32 OX_COMMAND`	`int32 SM_mathcap`

   Stack after the request:

`int32 OX_DATA`	*Mathcap* mathCapOb

   Output: none.

3. `SM_setMathcap`

   It requests a server to register the peer's mathcap `m` in the server. The server can avoid to send OX messages unknown to its peer.

   @plugin/mathcap.h)

Stack before the request: $\boxed{\textit{Mathcap m}}$

Request:

int32 OX_DATA	*Mathcap* m
int32 OX_COMMAND	int32 SM_setMathcap

Output: none.

Remark: In general the exchange of mathcaps is triggered by a client. A client sends SM_mathcap to a server and obtains the server's mathcap. Then the client registers the mathcap. Finally the client sends its own mathcap by SM_setMathcap and the server registers it.

4. SM_executeStringByLocalParser

   It requests a server to pop a character string s, to parse it by the local parser of the stack machine, and to interpret by the local interpreter. If the execution produces a Output, it is pushed onto OperandStack. If an error has occurred, Error2 Object is pushed onto the stack. OpenXM does not provide standard function names. If this operation and SM_popString is implemented, the stack machine is ready to be used as an OX server.

   Stack before the request:
   $\boxed{\textit{String commandString}}$

   Request:

int32 OX_COMMAND	int32 SM_executeStringByLocalParser

   Output: none.

   Remark: Before this request, one has to push *String commandString* onto the stack. It is done by sending the following OX data message.

int32 OX_DATA	int32 CMO_string	*size and the string commandString*

5. SM_executeStringByLocalParserInBatchMode

   This is the same request as SM_executeStringByLocalParser except that it does not modify the stack. It pushes an Error2 Object if an error has occurred.

6. SM_popString

   It requests a server to pop an object from OperandStack, to convert it into a character string according to the output format of the local system, and to send the character string via TCP/IP stream. (char *)NULL is returned when the stack is empty. The returned string is sent as a CMO string data. CMO_ERROR2 should be returned if an error has occurred.

   Stack before the request: $\boxed{\textit{Object}}$

   Request:

int32 OX_COMMAND	int32 SM_popString

   Output:

int32 OX_DATA	int32 CMO_STRING	*size and the string s*

7. SM_getsp

   It requests a server to push the current stack pointer onto the stack. The stack pointer is represented by a non-negative integer. Its initial value is 0 and a push operation increments the stack pointer by 1.

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`

   It requests a server to push a list object containing all error objects on the stack.

   Request: | int32 OX_COMMAND | int32 SM_dupErrors |

   Stack after the request: | int32 OX_DATA | *CMObject* a list of errors |

   Output: none.

9. `SM_pushCMOtag`

   It requests a server to push the CMO tag of the top object on the server stack. The tag is pushed as `CMO_INT32`. The top object remains on the stack. If there is no way to translate the object into CMO, push an error object.

   Request: | int32 OX_COMMAND | int32 SM_pushCMOtag |

   Stack after the request: | int32 OX_DATA | *CMO_INT32* tag |

   Output: none.

5.1.2 MathCap

Example: `ox_sm1` returns the following data as its mathcap.

```
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 ]]]
```

A mathcap has three components. The first one, which is also a list, contains informations to identify the version number of the OpenXM protocol, the system and hosts on which the application runs. In the above example, `Ox_system` denotes the system name. `HOSTTYPE` represents the OS type and taken from `$HOSTTYPE` enviroment variable. The second component consists of avaiable SM commands. The third component is a list of pairs. Each pair consists of an OX message tag and the list of available message tags. Again in the above example, 514 is the value of `OX_DATA` and it indicates that the server accepts CMO (without size information) as mathematical data messages. In this case the subsequent list represents available CMO tags.
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 Examples

Example: We show examples of `message_body`. Serial numbers are omitted.

1. `executeStringByLocalParser("12345 ;");`

   is converted into the following packet. Each number denotes one byte in hexadecimal representation. `(yy)` in `xx(yy)` represents the corresponding ASCII code.

   ```
   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
```

Each data has the following meaning.

```
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)
```

This is expressed by OXexpression as follows.

(OX_DATA, (CMO_STRING, 7, "12345 ;"))

(OX_COMMAND, (SM_executeStringByLocalParser))

2. A message which requests `SM_popString`:

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

In OXexpression it is represented as (OX_COMMAND, (SM_popString)).

The server returns the following reply message:

```
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)
```

In OXexpression it is represented as (OX_DATA, (CMO_STRING, 7, "12345 ;")).

5.1.4 Operators in the group SMobject/Basic

1. `SM_pops`

It requests a server to pop n and to discard elements *obj1, obj2, …, objn* from the stack.

Stack before the request:

obj1	*obj2*	⋯	*objn*	*Integer32 n*

Request:

`int32 OX_COMMAND`	`int32 SM_pops`

Output: none.

2. `int SM_setName`

   It requests a server to pop *name*, to pop *obj*, and to bind *obj* to a variable *name* in the current name space of the server. If an error has occurred `CMO_ERROR2` is pushed onto the stack.

   Stack before the request:

obj	*String name*

   Request:

`int32 OX_COMMAND`	`int32 SM_setName`

   Output: none.

3. `SM_evalName`

   It requests a server to pop *name* and to evaluate a variable *name* in the current name space. The Output of the evaluation *OutputObj* is pushed to the stack. If an error has occurred `CMO_ERROR2` is pushed onto the stack.

   Stack before the request:

String name

   Request:

`int32 OX_COMMAND`	`int32 SM_evalName`

   Stack after the request:

OutputObj

   Output: none.

4. `SM_executeFunction`

   It requests a server to pop s as a function name, to pop n as the number of arguments and to execute a local function s with n arguments popped from the stack. If an error has occurred `CMO_ERROR2` is pushed to the stack.

   Stack before the request:

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

   Request:

`int32 OX_COMMAND`	`int32 SM_executeFunction`

   Stack after the request: The Output of the execution.

   Output: none.

5. `SM_popSerializedLocalObject`

   It requests a sever to pop an object, to convert it into a serialized form according to a local serialization scheme, and to send it to the stream as an OX message. An OX message tag corresponding to the local data format must be sent prior to the serialized data itself. This operation is used mainly on homogeneous distributed systems.

6. `SM_popCMO`

   It requests a server to pop an object from the stack, to convert it into a serialized form according to the standard CMO encoding scheme, and to send it to the stream with the `OX_DATA` header.

   Request:

`int32 OX_COMMAND`	`int32 OX_popCMO`

   Output:

`int32 OX_DATA`	*Serialized CMO*

7. `SM_executeFunctionWithOptionalArgument`

   It requests a server to pop s as a function name, to pop an optional argument *opt*, to pop n as the number of arguments and to execute a local function s with n arguments popped from the stack. If an error has occurred `CMO_ERROR2` is pushed to the stack. *opt* is a list of lists of a keyword and a value. Keywords are strings.

   Stack before the request:

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

   Request:

`int32 OX_COMMAND`	`int32 SM_executeFunctionWithOptionalArgument`

   Stack after the request: The Output of the execution.

   Output: none.

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

   [Added in 2004-3-8]

6 Projects in work in progress

6.1 OX DATA with Length

`OX_DATA_WITH_LENGTH` is the OX tag for OX data message with a digital signature. It is followed by the serial number, CMO, an end mark and a digital signature. This type of OX data message is called *secured OX DATA*.

```
#define OX_SECURED_DATA              521
```

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

If `size` is equal to -1, then it is ignored.

tail is defined as follows.

`int32 CMO_START_SIGNATURE`	int32 size	*signature*

Here `size` is the length of *signature*. *signature* is a digital signature of *CMO data* by a Hash function and is used to detect invalid serialized objects. If `size` of Tail is equal to 0, then it has no digital signature.

Currently there are four modes of communicating data.

1. Only `OX_DATA` is used with checking by mathcap.
2. Only `OX_SECURED_DATA` is used.
3. Both `OX_DATA` and `OX_SECURED_DATA` can be used.
4. Only `OX_DATA` is used without checking by mathcap.

Suppose that the mathcap handling is incomplete and an application has received unknown CMObject. In mode 1, the application cannot detect the end of the CMObject and it will not be able to understand the subsequent messages. In mode 2, the application can detect the end of the unknown CMObject from the size information. However, in mode 2, additional cost is required on the sender to compute the total length of CMObjects.

Note that the exchange of mathcaps are not necessary at the start of a session. Any server should be implemented so that it can change the communication mode dynamically, say, from 4 to 1.

6.2 Local extension on server stack machines

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

0x10000 ID's beginning from `CMO_PRIVATE` = `OX_PRIVATE` = `SM_PRIVATE` are reserved for private use. They can be used to represent OX tags, CMObjects, SMobjects which are not authorized yet.

6.3 Implementation of other protocols such as MathLink and OpenMath

If we provide a library or a server for protocol conversion between CMO and "foreign" protocols such as MathLink or OpenMath, a client conforming to such protocols can communicate with Asir or kan/sm1 without knowing their internal structures.

6.4 Common operations on stack machines

Fundamental operations such as `add`, `sub`, `mul` should be executed on any server by `SM_executeFunction`. Control structures on stack machines such as `if` and `for` are also being considered.

7 Session Management

7.1 Control server

In OpenXM we adopted the following simple and robust method to control servers.

An OpenXM server has logically two I/O channels: one for exchanging data for computations and the other for controlling computations. The control channel is used to send commands to control execution on the nserver. The sample server (`oxmain.c`) processes such control messages on another process. We call such a process a *control server*. In contrast, we call a server for computation an *engine*. As the control server and the engine runs on the same machine, it is easy to send a signal from the control server. A control server is also an OpenXM stack machine and it accepts `SM_control_*` commands to send signals to a server or to terminate a server.

7.2 New OpenXM control servers

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

7.3 OpenXM reset protocol

A client can send a signal to an engine by using the control channel at any time. However, I/O operations are usually buffered, which may cause troubles. To reset an engine safely the following are required.

1. Any OX message must be a synchronized object in the sense of Java.
2. After restarting an engine, a request from a client must correctly corresponds to the response from the engine.

`SM_control_reset_connection` is a stack machine command to initiate a safe resetting of an engine. The control server sends `SIGUSR1` to the engine if it receives `SM_control_reset_connection` from the client. Under the OpenXM reset protocol, an engine and a client act as follows.

Client side

1. After sending `SM_control_reset_connection` to the control server, the client enters the resetting state. It discards all `OX` messages from the engine until it receives `OX_SYNC_BALL`.
2. After receiving `OX_SYNC_BALL` the client sends `OX_SYNC_BALL` to the engine and returns to the usual state.

Engine side

1. After receiving `SIGUSR1` from the control server, the engine enters the resetting state. The engine sends `OX_SYNC_BALL` to the client. The operation does not block because the client is now in the resetting state.
2. The engine discards all OX messages from the engine until it receives `OX_SYNC_BALL`. After receiving `OX_SYNC_BALL` the engine returns to the usual state.

Figure 1 illustrates the flow of data. `OX_SYNC_BALL` is a special OX message and is used to mark the end of data remaining in the I/O streams. After reading it, it is assured that each stream is empty and that the subsequent request from a client correctly corresponds to the response from the engine.

7.4 Control message (SMObject/TCPIP/Control)

1. `SM_control_reset_connection`

   It requests a control server to send `SIGUSR1` to the engine. The control server should immediately reply an acknowledgment to the client.

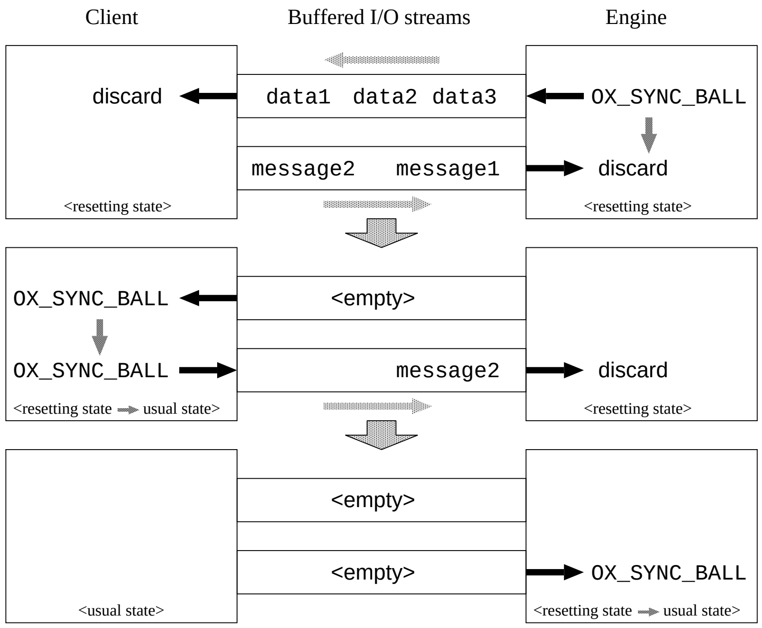

Figure 1: OpenXM reset procedure

Request: | `int32 OX_COMMAND` | `int32 SM_control_reset_connection` |

Result: | `int32 OX_DATA` | `CMO_INT32` result |

2. `SM_control_kill`

   It requests a control server to terminate the engine and the control server itself. All files and streams should be closed before the termination of servers.

   Request: | `int32 OX_COMMAND` | `int32 SM_control_kill` |

   Result: none.

Example: (serial numbers are omitted.)

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

Reply to the reset request

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

`OX_SYNC_BALL` are exchanged on the data channel for synchronization.

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

7.5 Notification from servers

OpenXM servers try to be as quiet as possible. For example, engine errors of a server are only put on the engine stack and the engine does not send error packets unless the client sends the message `pop_cmo`.

OpenXM provides a method to notify events. Control server may send `OX_NOTIFY` header and an `OX_DATA` packet. This transmission may be prohibited by mathcap.

Let us explain how to use `OX_NOTIFY` by an example. The `ox_plot` server of `asir` has a quit button. If the quit button is pressed, the canvas dissappears, but the engine does not terminate. If the client sends drawing messages without the canvas, then the engine pushes error packets "canvas does not exist" on the engine stack. If the engine wants to notify the error to the client immediately, the `OX_NOTIFY` message should be used.

Let us note that only the control process is allowed to send `OX_NOTIFY`. Therefore, the engine must ask the control server to send `OX_NOTIFY`. Methods to ask the control process from the engine depends on operating system. In case of unix, one method is the use of a file; for instance, if the engine touches the file `/tmp/.ox_notify.pid`, then the control server sends the `OX_NOTIFY` header and the `OX_DATA` packet of `cmo_null`. Here, `pid` is the process id of the engine. Engines and control processes may use a shared memory or a signal instead of the file `/tmp/.ox_notify.pid`.

8 How to start a session on TCP/IP

8.1 Standard I/O on OX servers

In order to make it easy to implement servers, one can assume that any server has two opened socket descriptors 3 and 4, which are for input from a client and for output to a client respectively. That is, servers do not have to do socket operations to establish connections. However servers are responsible for buffering data to exchange OX messages efficiently. Note that associating a buffered stream with a descriptor can be done by `fdopen()`.

8.2 Launcher

Though there need several socket operations to establish a connection over TCP/IP, servers do not have any functionality for connection establishment. An application called *launcher* is provided to start servers and to establish connections as follows.

1. A launcher is invoked from a client. When the launcher is invoked, the client informs the launcher of a port number for TCP/IP connection and the name of a server.

2. The launcher and the client establish a connection with the specified port number. One time password may be used to prevent launcher spoofing.

3. The launcher creates a process and establishes a connection to the client. Then the launcher arranges for the newly created descriptors to be 3 and 4, and executes the specified server.

After finishing the above task as a launcher, the launcher process acts as a control server and controls the server process created by itself.

8.3 Negotiation of the byte order

A client and a server exchange one byte data soon after the communication has started as follows.

- The server writes one byte representing the preferable byte order to the client, then waits for one byte to come from the client.
- After reading the byte, the client writes one byte representing the preferable byte order to the server.

The one byte data is `0`, `1` or `0xFF`. `0` means that one wants to use the network byte order to send 16 or 32bit quantities. `1` means that one wants to use the little endian order. `0xFF` means that one wants to use the big endian order. On each side, if the preference coincides with each other then the byte order is used. Otherwise the network byte order is used.

If a system implements only the network byte order, then it is sufficient to send always `0`. However unnecessary byte order conversion may add large overhead and it is often a bottle-neck on fast networks.

In order to send and receive 64 bit machine double (floating point number) and 128 bit machine double, we use the same byte order. In other words, we cast `double64 *` to `int32 *` and send the array of 4 bytes by the same method with sending `int32`. As to examples, see the section on `CMO_64BIT_MACHINE_DOUBLE`.

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

8.4 An example of launcher : `ox`

`ox`, included in `OpenXM/src/kxx`, is a launcher to invoke an engine. After invoking an engine, it acts as a control server. By default `ox` requires a one time password. To skip it, use `-insecure` option. A one time password is a null-terminated byte sequence and a client informs both a control server and an engine of byte sequences as one time passwords.

`ox` is created from `oxmain.c` and `kan96xx/plugin/oxmisc.c`. In `ox` `oxTellMyByteOrder()` executes the exchange of the byte order information. In a client it is done in `oxSetByteOrder()`.

One time passwords should be sent via secure communication channels. Note that in the current implementation of `ox`, one time passwords are visible to all users logging in machines on which the server and the client run, assuming that there is no evil person among the users. One may use `ssh` with `-f` option when one wants to send a one time password securely to a remote machine.

The following example shows invocation of an `ox_sm1` server and the communication establishment on sm1. In this example `ox` on the host `dc1` is invoked from `sm1` on the host `yama`.

```
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 Example of using OX servers

An sample C source code to use ox servers by TCP/IP can be found in `OpenXM/doc/oxlib/test1-tcp.c`.

9 String expression of objects

Objects may be serialized as a string. The string expression of an object of the system xxx is accepted as a string expression for the OX xxx server.

10 CMOexpressions for numbers and polynomials

@../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 */
```

In the sequel, we will explain on the groups CMObject/Basic, CMObject/Tree and CMObject/DistributedPolynomial.
The program `bconv` at `OpenXM/src/ox_toolkit` translates CMO expressions into binary formats. It is convinient to understand the binary formats explained in this section.
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 $\in$ CMObject/Basic.

Zero	:	(`CMO_ZERO`)
		— Universal zero
ZZ	:	(`CMO_ZZ`, $int32$ f, $byte$ a[1], $\ldots$, $byte$ a[|m|])
	:	— bignum. The meaning of a[i] will be explained later.
QQ	:	(`CMO_QQ`, $int32$ m, $byte$ a[1], $\ldots$, $byte$ a[|m|], $int32$ n, $byte$ b[1], $\ldots$, $byte$ b[|n|])
		— Rational number a/b.
Rational	:	(`CMO_RATIONAL`, $CMObject$ a, $CMObject$ b)
		— Rational expression a/b.
Indeterminate	:	(`CMO_INDETERMINATE`, $Cstring$ v)
		— Variable name v .

The name of a variable should be expressed by using Indeterminate. v may be any sequence of bytes, but each system has its own restrictions on the names of variables. Indeterminates of CMO and internal variable names must be translated in one-to-one correspondence.

10.1 Indeterminate and Tree

Group CMObject/Tree requires CMObject/Basic.
Tree, Lambda $\in$ CMObject/Tree.

Tree : ($\mathtt{CMO\_TREE}, Cstring\,\mathrm{name}, List\,\mathrm{attributes}, List\,\mathrm{leaves}$)
— "name" is the name of the node of the tree.
— Attributes may be a null list. If it is not null, it is a list of
— key and value pairs.

Lambda : ($\mathtt{CMO\_LAMBDA}, List\,\mathrm{args}, Tree\mathrm{body}$)
— a function with the arguments body.

In many computer algebra systems, mathematical expressions are usually expressed in terms of a tree structure. For example, $\sin(x + e)$ is expressed as `(sin, (plus, x, e))` as a tree. Tree may be expressed by putting the expression between `SM_beginBlock` and `SM_endBlock`, which are stack machine commands for delayed evaluation. (cf. { , } in PostScript). However it makes the implementation of stack machines complicated. It is desirable that CMObject is independent of OX stack machine. Therefore we introduce an OpenMath like tree representation for CMO Tree object. This method allows us to implement tree structure easily on individual OpenXM systems. Note that CMO Tree corresponds to Symbol and Application in OpenMath.

Lambda is used to define functions. The notion "lambda" is borrowed from the language Lisp.
Example: the expression of $sin(x + e)$.

```
(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"),  the base of natural logarithms
    (CMO_LIST,[size=]1,(CMO_LIST,[size=]2,(CMO_STRING, "cdname"),
                                          (CMO_STRING,"basic")))
        ))
    )
)
```

Elements of the leave may be any objects including polynomials.
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>
```

Let us define a group for distributed polynomials. In the following, DMS stands for Distributed Monomial System.

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

Monomial	:	Monomial32 | Zero
Monomial32	:	(`CMO_MONOMIAL32`, $int32\,n$, $int32\,\mathrm{e}[1], \ldots, int32\,\mathrm{e}[\mathrm{n}]$, Coefficient)
		— e[i] is the exponent e_i of the monomial $x^e = x_1^{e_1} \cdots x_n^{e_n}$.
Coefficient	:	ZZ|Integer32
Dpolynomial	:	Zero
		| (`CMO_DISTRIBUTED_POLYNOMIAL`, $int32\,m$, DringDefinition, [Monomial32|Zero], {Monomial32})
		— m is equal to the number of monomials.
DringDefinition	:	DMS of N variables
		| RingByName
		| Generic DMS ring
		— definition of the ring of distributed polynomials.
Generic DMS ring	:	(`CMO_DMS_GENERIC`)
RingByName	:	(`CMO_RING_BY_NAME`, $Cstring\,s$)
		— The ring definition referred by the name "s".

DMS of N variables : (`CMO_DMS_OF_N_VARIABLES`,
(`CMO_LIST`, $int32\,\mathrm{m}$, $Integer32\,\mathrm{n}$, $Integer32\,\mathrm{p}$
$[, Cstring\,\mathrm{s}, List\,\mathrm{vlist}, List\,\mathrm{wvec}, List\,\mathrm{outord}])$
— m is the number of elements.
— n is the number of variables, p is the characteristic
— s is the name of the ring, vlist is the list of variables.
— wvec is the weight vector.
— outord is the order of variables to output.

Note that it is possible to define DMS without RingByName and DMS of N variables.

In the following we describe how the above CMObjects are implemented on Asir and Kan.

10.2 Zero

Note that CMO has various representations of zero.

10.3 Integer ZZ

```
#define     CMO_ZZ          20
```

We describe the bignum (multi-precision integer) representation `CMO_ZZ` in OpenXM. The format is similar to that in GNU MP. (cf. `plugin/cmo-gmp.c` in the `kan/sm1` distribution). CMO_ZZ is defined as follows.

`int32 CMO_ZZ`	`int32` f	`int32` b_0	$\cdots$	`int32` b_n

f is a 32bit integer. $b_0, \ldots, b_n$ are unsigned 32bit integers. $|f|$ is equal to $n+1$. The sign of f represents that of the above integer to be expressed. As stated in Section 2, a negative 32bit integer is represented by two's complement.

In OpenXM the above CMO represents the following integer. ($R = 2^{32}$.)

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

Example: If we express `int32` by the network byte order, a CMO_ZZ 14 is expressed by

$$(\mathrm{CMO\_ZZ},\ 1,\ 0,\ 0,\ 0,\ \mathrm{e}),$$

The corresponding byte sequence is

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

Note that CMO_ZZ 0 is expressed by `(CMO_ZZ, 00,00,00,00)`.

10.4 Distributed polynomial Dpolynomial

We treat polynomial rings and their elements as follows.

Generic DMS ring is an n-variate polynomial ring $K[x_1, \ldots, x_n]$, where K is a coefficient set. K is unknown in advance and it is determined when coefficients of an element are received. When a server has received an element in Generic DMS ring, the server has to translate it into the corresponding local object on the server. Each server has its own translation scheme. In Asir such an element are translated into a distributed polynomial. In `kan/sm1` things are complicated. `kan/sm1` does not have any class corresponding to Generic DMS ring. `kan/sm1` translates a DMS of N variables into an element of the CurrentRing. If the CurrentRing is n'-variate and $n' < n$, then an n-variate polynomial ring is newly created.

If RingByName (`CMO_RING_BY_NAME`, yyy) is specified as the second field of DMS, it requests a sever to use a ring object whose name is yyy as the destination ring for the translation.

Example: (all numbers are represented in hexadecimal notation)

```
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),),
```

$3x^2y$ is regarded as an element of a six-variate polynomial ring.

10.5 Recursive polynomials

```
#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 $\in$ CMObject/RecursivePolynomial

Polynomial in 1 variable : (`CMO_POLYNOMIAL_IN_ONE_VARIABLE`, *int32* m,

		Name of the main variable ,
		{ *int32* e, Coefficient })
		— m is the number of monomials.
		— A pair of e and Coefficient represents a monomial.
		— The pairs of e and Coefficient are sorted in the decreasing order, usually with respect to e.
		— e denotes an exponent of a monomial with respect to the main variable.
Coefficient	:	ZZ | QQ | integer32 | Polynomial in 1 variable | Tree | Zero | Dpolynomial
Name of the main variable	:	*int32* v
		— v denotes a variable number.
Recursive Polynomial	:	(`CMO_RECURSIVE_POLYNOMIAL`, RringDefinition, Polynomial in 1 variable | Coefficient)
RringDefinition	:	*List* v
		— v is a list of names of indeterminates or trees.
		— It is sorted in the decreasing order.

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)))
```

This represents

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

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

10.6 CPU dependent 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 $\in$ CMObject/MachineDouble

64bit machine double	:	(`CMO_64BIT_MACHINE_DOUBLE`,
		byte s1 , . . . , *byte* s8)
		— s1, . . ., s8 `double` (64bit).
		— Encoding depends on CPU.
		Need the byte order negotiation.
Array of 64bit machine double	:	(`CMO_ARRAY_OF_64BIT_MACHINE_DOUBLE`, *int32* m,
		byte s1[1] , . . . , *byte* $s8[1], \ldots,$ *byte* $s8[m]$)
		— s*[1], . . . s*[m] are 64bit double's.
		— Encoding depends on CPU.
		Need the byte order negotiation.
128bit machine double	:	(`CMO_128BIT_MACHINE_DOUBLE`,
		byte s1 , . . . , *byte* s16)
		— s1, . . ., s16 `long double` (128bit).
		— Encoding depends on CPU.
		Need the byte order negotiation.
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] are 128bit long double's.
		— Encoding depends on CPU.
		Need the byte order negotiation.

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

See IEEE 754 double precision floating-point (64 bit) for the details of float compliant to the IEEE standard.

The internal expression of 256.100006 in the Intel Pentium is `cd 0c 80 43` The internal expression of 256.100006 in the PowerPC (Mac) is `43 80 0c cd` . As you have seen in this example, the orders of the bytes are opposite each

other. The byte order is specified by the byte order negotiation protocol when the engine starts.
Group CMObject/Bigfloat requires CMObject/Primitive, CMObject/Basic.
Bigfloat $\in$ CMObject/Bigfloat

$$\begin{array}{rcl} \text{Bigfloat} & : & (\texttt{CMO\_BIGFLOAT}, \\ & & \quad \mathit{ZZ}\ \text{a}\ ,\ \mathit{ZZ}\ \text{e}) \\ & & \text{---}\ a \times 2^e. \end{array}$$

11 OX Local Data

Each serialization scheme of object proper to a system is tagged with a number greater than or equal to

```
#define   OX_LOCAL_OBJECT       0x7fcdef30
```

There are 0x200000 rooms for such tags. A tag is supplied for each system.

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

12 CMO ERROR2

Error numbers (common to all systems)

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

13 Registering a new CMO

13.1 Requirement for a new CMO

CMO data types are defined recursively. Thus, if one introduces a new CMO, then old CMO's may be also extended.

13.2 How to join in the OpenXM project

You are welcome to add packages to OpenXM. Ask takayama@math.kobe-u.ac.jp for details. You may introduce new CMO's if necessary. If you have defined a new CMO, send

1. the formal definition and an explanation of the CMO,

2. an explanation of the behavior of a system xxx for the CMO,
3. URL's related to the CMO or xxx.

After discussing on the new CMO, we will fix the specification. Then we will issue the tag for the new CMO and create links to the URL related to the CMO from the OpenXM home page [5].

14 OX servers as a C library

In some OX servers, one can use the OX server as a C library. The API to the C library is similar to Asir OX client API such as `ox_push_cmo()`, `ox_pop_cmo()`.

CMO should be converted into the binary encoded form to call these functions.

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

This function initializes the library interface. `type` specifies the byte order to send int32 to the OX server xxx. If type is equal to 0, the native byte order will be used. If type is equal to 1, the network byte order will be used. In case of error, -1 will be returned.

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

Push the binary encoded CMO `cmo` onto the stack of the OX server xxx.

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

Pop a binary encoded CMO from the OX server xxx and write it at `cmo`. The return value is the size of the CMO in bytes. In case of the stack underflow, the return value is 0. If the size exceeds the `limit`, -1 will be returned and the CMO is not popped and will not be written to `cmo`.

```
int xxx_ox_peek_cmo_size()
```

Return the size of the CMO at the top of the stack.

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

This function sends a stack machine command (`OX_COMMAND`,int32 cmd) to a server.

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

This function requests a server to execute a command expressed by a string `s`. `s` should be acceptable by the parser of the server.

Sample source codes to use the library mode interface can be found in `OpenXM/doc/oxlib`.

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`